Introduction

Motovation

The Rice Robotics Club (RRC) is attempting to build an autonomous robot to maneuver around the Rice beer bike track and eventually all of campus. To achieve this, the robot will need to be able to identify and follow traffic laws. To help the Robotic Club with their task, we explored street sign detection methods using two common image processing techniques: template matching and edge detection. Ideally, exploration of these methods would allow the creation of an image processing algorithm that can identify relevant signs as well as note other information such as position, distance, and content for each sign observed.

This module explains and compares three methods to detect signs: one based purely on template matching, one based purely on edge detection, and another that uses both template matching and edge detection. Most of our work in this study concentrated on stop signs and speed limit signs, those being the signs most commonly found on campus, but we also explored one way signs and do not enter signs to a lesser extent.

Background

Template Matching

Template matching involves comparing regions of an image to a template, using correlation algorithms to detect regions that are similar. Correlation is similar to convolution but without flipping one of the signals. For use in image processing, we must normalize the signals to account for variations of intensity in the image matrices, one for each color matrix (red, green, and blue). Using the MATLAB function **normxcorr2**, we are able to take a normalized 2-D cross-correlation of two images, with the regions most similar to the template in the picture returning the highest correlation values.

Examining a cross-correlation formula we see the similarity to convolution except neither image is flipped:

$$(f \star g)[n] \overset{\text{def}}{=} \sum_{m=-\infty}^{\infty} f^*[m]\ g[n+m].$$

In order to expand to two dimensions and normalize the correlation, we must use linear algebra. Since MATLAB allows us to convert images to 2-D matrices, three for a color image (the RGB values at each x,y coordinate pair), we can run a similar formula but use vectors instead of points. The normalization factor on the bottom scales the values so they do not vary based on the high intensities of individual colors in parts of the image. Without this correction, a bright sky would always have high correlation with the image just because it is bright.

$$\gamma(u,v) = \frac{\sum_{x,y}\left[f(x,y)-\bar{f}_{u,v}\right]\left[t(x-u,y-v)-\bar{t}\right]}{\left\{\sum_{x,y}\left[f(x,y)-\bar{f}_{u,v}\right]^2\sum_{x,y}\left[t(x-u,y-v)-\bar{t}\right]^2\right\}^{0.5}}$$

Thus, using a stop sign template we can attempt to match it with an image and extract the location of highest correlation. At this point, we can either check it against a threshold value to determine if it matches, or do further analysis on regions of the image.

Edge Detection

Edge detection involves analyzing changes in intensity to determine the edges in an image, and then further analysis can be performed on certain bounded regions of the image. Given an colored image, MATLAB has the capability to convert it to a 2-D grayscale image using **rgb2gray**. Then, we can capitalize on the changes in intensity to detect edges. The MATLAB algorithm we chose to accomplish this is the Sobel method, which approximated the gradient of the image using a convolution with the following matrices.

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

The regions with the highest changes in intensity are the most likely to be edges in an image. Given the probable edges, we can use the function **bwareaopen** to remove any bounded regions that are too small to be of use in our image processing. Left with only large boundaries, we can now test each one for various features such as color, shape, area, etc. to determine if it is a street sign.

Approaches

Approach I: Template Matching Only

In the template matching algorithm, a sign template is correlated with a larger image that may or may not include the sign. We used the MATLAB function **normxcorr2** to correlate the template and image. Though template matching works perfectly when a smaller reference image is directly cropped from the larger image, in practice we would not have the cropped image. Thus, our first task was to choose a general reference image. After finding a good reference image, our second task was to write a program that would find the correlation for many image:reference ratios which would then allow us to estimate the size of the sign within the larger image. This is necessary because we don't know the size of the stop sign in the larger image and if the template is the wrong size there will not be a high correlation even if the template is in the image.

We implemented the template matching algorithm on stop signs first. The first template tried for stop signs was stop signs with a black background (fig. 1). This template yielded results of ~75% stop signs detected. We saw improved results, ~85% signs detected, if instead of using a template with the entire sign, we used a template that was a rectangular red box with the word “stop” (fig. 2). This is because the area around the edge of the stop sign in the image was significantly different than the template with a purely black background leading to lower correlation values.

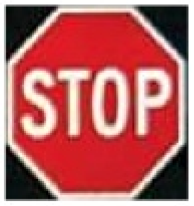

Fig. 1 First template used to detect stop signs

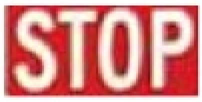

Fig. 2 Second template used to detect stop signs

When a full speed limit sign with a single speed was used for the general template, we lost accuracy because the difference in the speed limits themselves (i.e. 55 vs. 40) reduced the correlation. The top half of a speed limit sign (fig. 3) proved to be a much better template and we were able to detect a majority of speed limit signs (~85%). We were also able to successfully implement our code on do not enter and one way signs.

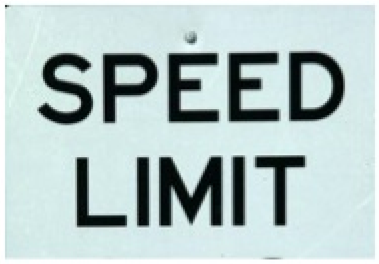

Fig. 3 Template used to detect speed limit signs

If a template is a different size than the sign within the larger image, the sign will not be detected. To solve this problem we calculated the correlation of the template and image for many different template/image ratios. We scaled down the image as oppose to scaling up the template to cut down on computation time. When a sign is present and when the template and image have the correct ratio, there is a spike in the correlation values (fig. 4). When no sign that matches the template is present, there is not spike in correlation (fig. 5). By looking at the scaling of highest correlation we are able to estimate the size of the sign and by looking at the point of highest correlation for this ratio we are able to determine the location of the sign.

Correlation as Image Size is Scaled

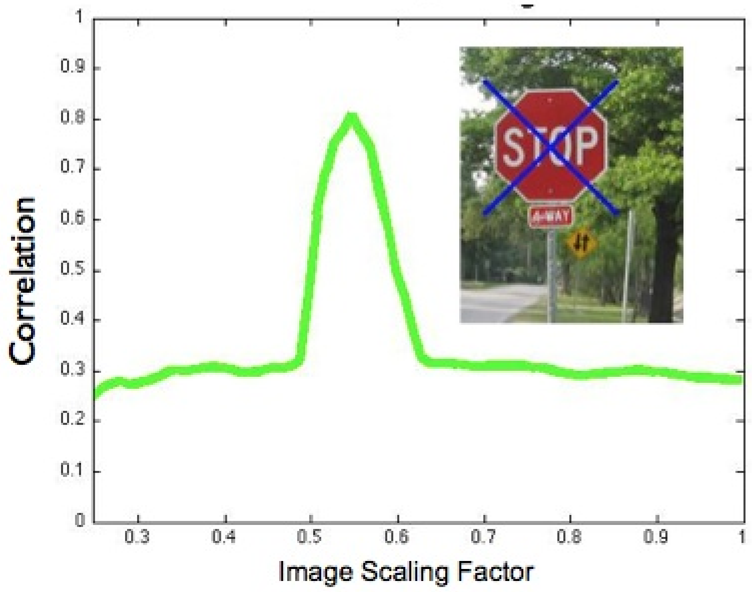

Fig. 4 The image size is scaled as the reference size remains constant. As the image:reference ratio changes, the correlation changes reaching a maximum when the reference is close in size to the size in the image The x marks the spot of highest correlation and where our program thinks a stop sign is present

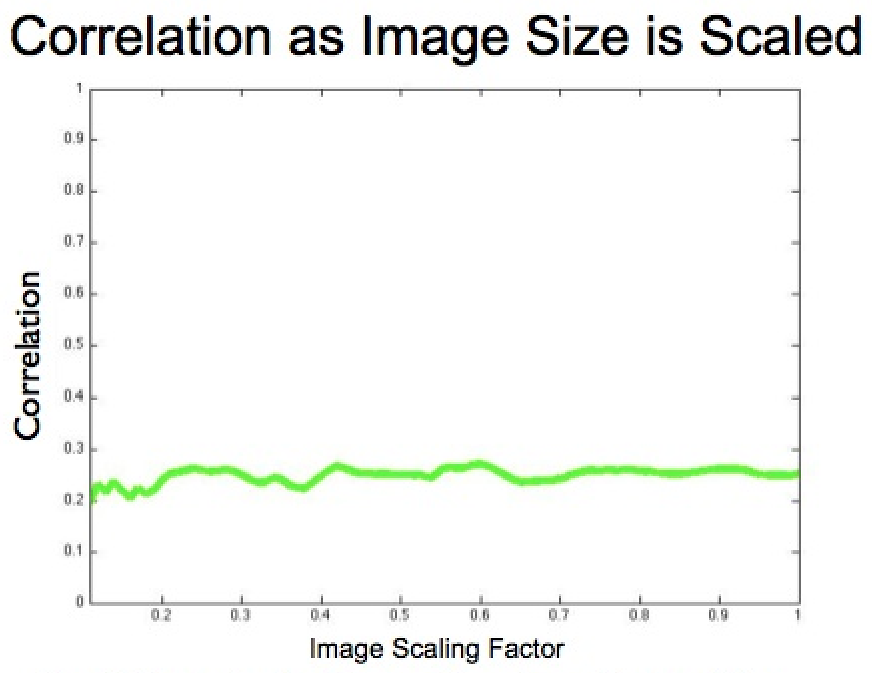

Fig. 5 When a sign is not present in an image, the correlation value never reaches a value above threshold, no matter matter how the image is scaled.

Sign	Reference	Percentage Above Threshold
Stop	Stop	90%
Speed Limit	Stop	0%
Do Not Enter	Stop	33%
One Way	Stop	66%

Approach II: Edge Detection Only

In the Edge Detection algorithm, regions of the image are found and then put through another algorithm that determines if they are a part of the sign. We used the MATLAB function **edge** to find the boundaries of the objects in the image (fig. 6) and then used the MATLAB function **regionprops** to find regions within the image.

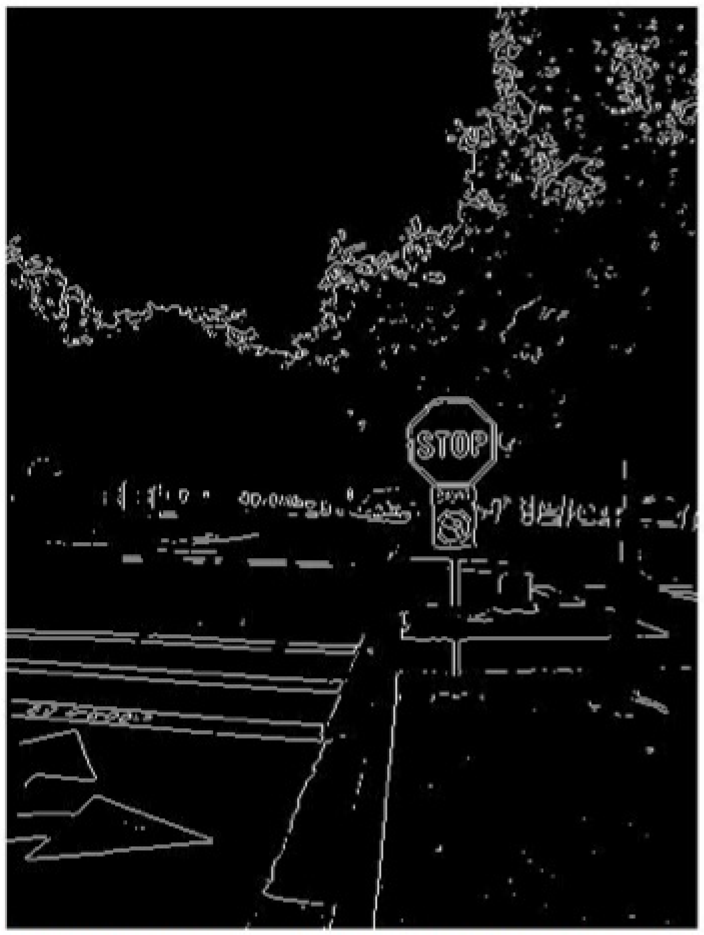

Fig. 6 After the image has been edge detected, only the outlines of objects show

We first tried to find the border of the sign. The outside edge of the sign is a region most of the time, so this was a possibility. However, there is not a good way of determining which region was the sign. A first guess was to choose the biggest region, however this did not always work as the sign was not always the biggest object in the picture. We then noticed that the O of stop showed up as two regions, the inner ring and the outer ring. Since these two regions have approximately the same center (fig. 7), it is possible to find regions that have centers close together and then analyze them to see if they are the sign in question. For example, if the point is on a stop sign, then the point, the point a bit above it, and the point a bit below it will all be red. If the point is on a do not enter sign, then the point will be white and the points above and below will be red.

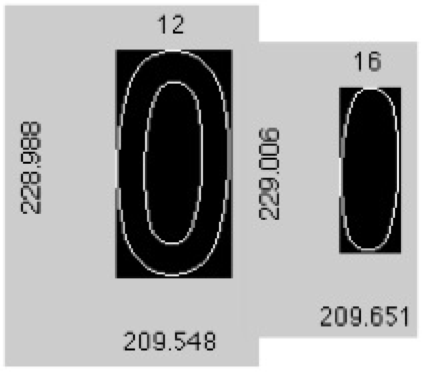

Fig. 7 Inside and outside of "O" have same center (axis titles are point values)

The points above and below were chosen by using the height of the region found. By choosing points the height of the region above and below the identified center, the points are on the outside of the letter. This tells the background of the sign itself.

This method of finding signs is not the most accurate. It can be fooled by other signs. For example, a sign for a business that had an O on a red background would be considered a stop sign by this algorithm. This could be helped by checking more points. Points immediately around the center should also be red and some points in between the center and the currently checked outer points should be white. This would improve the accuracy, but it could still be fooled. This algorithm also works best for signs with color. Signs with white backgrounds such as speed limit signs are hard to find because values that are defined as 'white' cover a large range of pale colors and therefore many things not signs are identified as signs.

Despite these drawbacks, this code is advantageous because it does not require large convolutions like the first method. This means that the method is much faster. To find the sign of the correct size with the first approach, it takes about 30 seconds. To find the sign including the size with the second approach, it takes about .5-1 second.

Sign	Reference	Percentage Above Threshold
Stop	Stop	95%
Speed Limit	Stop	0%
Do Not Enter	Stop	15%
One Way	Stop	0%

Approach III: Template Matching and Edge Detection

For our final algorithm we used both template matching and edge detection in an attempt to combine the positive aspects of the two approaches. The combined algorithm first utilizes edge detection to isolate regions within the image, throwing out all regions of a size below a certain threshold in order not to waste time with extraneous convolutions. Once the significant regions, usually numbering between twenty and one hundred, have been identified, each is re-sized to a standard small template size and compared to each image in the template library. Regions with high correlations with a template image are assumed to be the part of the sign that template was created from.

This combined algorithm performed quite well, although it was not error-free. As was the case with pure template matching, attempting to match pictures of entire signs was prone to error because the background of any given sign can vary, occasionally leading to quite low correlations. However, we discovered that using this algorithm to match only specific elements of a sign, such as the “S”, “T”, “O”, and “P” in stop (fig. 8), was much more reliable, as the conditions for such sub-sign features are much more consistent than for the overall sign. In addition, searching for individual sign features and not just the sign itself means that partially obscured signs that would be beyond our first template matching algorithm would present no problems for this modified one. Our template library was expanded so as to contain a number of sub-sign features for both stop signs and speed limit signs, as both of these signs contain changing characteristics (background for the stop sign, speed limit for the speed limit sign) that can thwart conventional template matching. Again, we found that matching individual letters yielded a high success rate (fig. 9). Our basic algorithm was also fairly successful in identifying one way signs and do not enter signs, even though no sub-feature library was created for these signs and only the sign itself was used as a template.

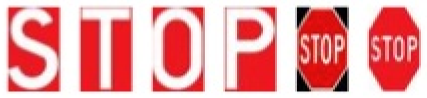

Fig. 8 Using individual letters was more accurate than using entire signs

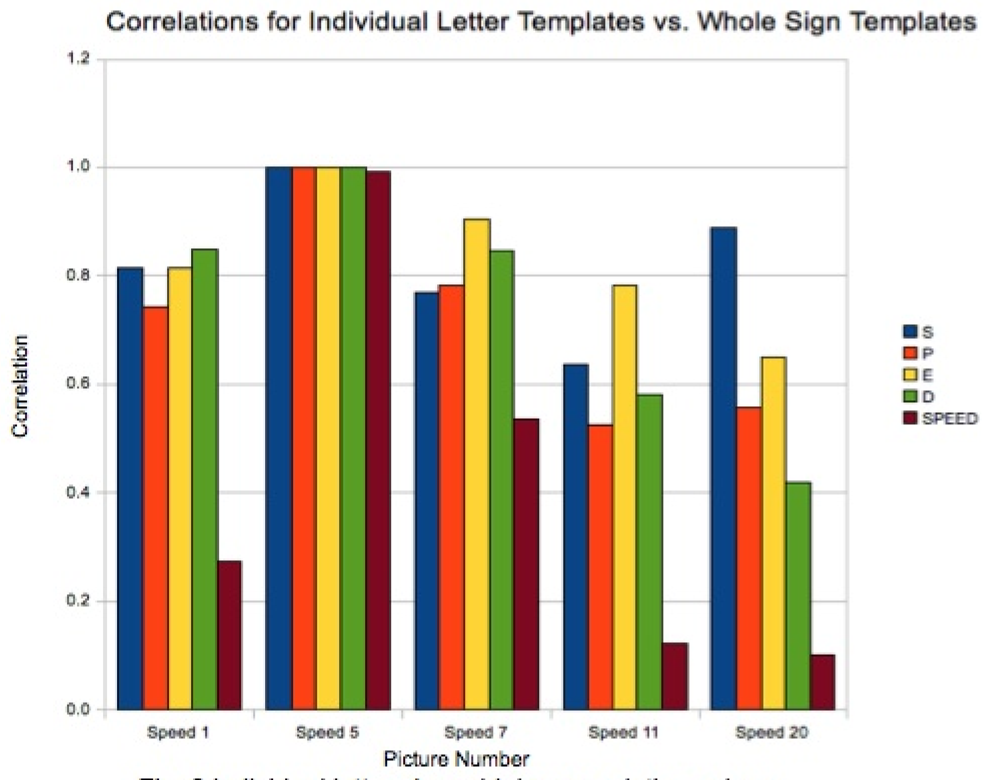

Fig. 9 individual letters have higher correlation values

This algorithm showed significant success in combining the positive aspects of both tablet matching and edge detection. In comparison to pure template matching, this method performs correlation calculations on much smaller images and size matching is a non-issue because all regions are re-sized to match the templates. In comparison to pure edge detection, this method is less prone to false positives and can be much more easily expanded to detect new signs simply by adding to the template library. Overall, the combination of template matching and edge detection yields a more robust, flexible, and reliable method of sign detection than either algorithm used alone.

Sign	Reference	Percentage Above Threshold
Stop	Stop	95%
Speed Limit	Stop	15%
Do Not Enter	Stop	66%
One Way	Stop	33%

Conclusions

Combining methods gives best performance

Image processing techniques can be used to not only detect street signs but also read data from them. Using MATLAB algorithms, techniques such as template matching, edge detection, and combinations of the two can be used to analyze an image for street signs, detecting the presence of the sign and position of it to a reasonable accuracy. Through our group’s work, we found that combining template matching and edge detection provided the best approach that can be adapted with clever algorithms to detect a variety of signs. Though work remains, these approaches will prove useful to the Rice Robotics Team in continuing their work on an unmanned vehicle.

Future Work

Future Improvements

As a result of this study, we believe that the third method, the combination of template matching and edge detection, displays the most promise, and so future work would focus on this method. The accuracy could be improved by adding more template images of each type of sign in order to detect non-standard angle and lighting conditions, and the number of sign types detected could be easily expanded by adding more template images to the library. Another expansion we considered would be reading specific useful information off of identified signs. For example, although our code does not currently read the speed off of speed limit signs, this would be easy to implement within the framework of the third method’s algorithm. When a speed limit sign is detected, nearby regions would be template matched with the numerical digits 0 through 9 and their relative positions would be used to determine the proper order, thus identifying the speed limit. Also, as the final end-use application we envision is real-time sign detection for the purposes of autonomous robot navigation, our code needs to be able to process a live video stream. This would require insuring that the chosen algorithm runs fast enough for a significant amount of frames to be analyzed.

Code

Apporach 1

```
function sign_detector_basic

color = [85 255 0]./255;


%%% READ IN WHICHEVER REFERENCES YOU WOULD LIKE TO
USE
stop_gen_ref_2 = double(imread('ref_stop.png'));
size_gen_ref = size(stop_gen_ref_2);




%%% INPUT IMAGE TO BE DETECTED
file_image = strcat('speed_limit6.jpg');
sign = double(imread(file_image));
size_sign = size(sign);

%%% c is the ratio of the image edges.  In order
to avoid distorting our
%%% image we want this ratio to stay constant as
we resize the image
c = size_sign(2)/size_sign(1);

%%% This logic determines which side is smaller.
The first image:reference
%%% ratio we try is 2:1 where the shortest side of
the image is twice the
%%% size of the reference.
n1 = (size_gen_ref(2)/size_sign(2));
n2 = size_gen_ref(1)/size_sign(1);
n = max(n1, n2);
```

```
max_chart = [];
max_val_plot = 0;

%%% This loop finds the correlation for
image:ratio sizes of n through 1
for i = n:.01:1

    %Resize Image
    yi = size_sign(1)*i;
    xi = c*yi;
    sign_res = imresize(sign, [yi xi]);

    %Call finder to determine where the point of
highest correlation is and
    %update the vector max_chart which stores the
maximum value for each
    %iteration
    [max_val, locx, locy] = finder(sign_res,
stop_gen_ref_2);
    max_chart = [max_chart; i max_val];

    %If the new max value is higher, record the
sign, location, and value
    %of the correlation for this ratio.
    if max_val > max_val_plot
        max_val_plot = max_val;
        locx_plot = locx;
        locy_plot = locy;
        sign_plot = sign_res;

    end


end

%Plot a figure that shows how the correlation
varies as the image:reference
```

```
%ratio varies
figure
plot(max_chart(:,1), max_chart(:,2), 'color',
color, 'LineWidth', 5);
sign_title = strcat('Correlation as
Image:Reference Ratio Varies');
title(sign_title)
axis([n, 1, 0 ,1])

%Plot a figure that shows where the program finds
the sign at the
%image:reference ratio that gives the maximum
correlation
figure
imshow(uint8(sign_plot));
hold on
plot(locy_plot, locx_plot, 'bx', 'MarkerSize',
100, 'LineWidth', 3)
title(strcat(file_image,' ', num2str(locx),' , ',
num2str(locy)));



end


%This funciton determines the maximum value of
correlation and location of
%this maximum value for an inputted imag and
reference.
function [max_val, locx, locy] =
finder(stop_image, reference)

%Find the sizes of the matrices
size_image = size(stop_image);
size_ref = size(reference);
```

```
%Grab first two columns
size_image = size_image(1:2);
size_ref = size_ref(1:2);
%Calculate amount for zero-padding
x = size_image + size_ref - [1 1];


correlation = zeros(x(1), x(2));
for k = 1:3
    %Grab values for one color
    stop_image_t = stop_image(:,:,k);
    reference_t = reference(:,:,k);
    %Calculate and add the correlations for each
color
    correlation_int = normxcorr2(reference_t,
stop_image_t);
    correlation = correlation + correlation_int;
end

correlation = correlation./3;

%Determine the max value
max_val = max(max(correlation));
%The correlation matrix is not the same size of
the image becuase the size
%of a signal resulting from a convolution has a
size equal to the sum of
%the sizes of the two images that were convolved.
Hence, need to determine
%which points in the matrix correlation refer to
which points in the image.
[x,y] = size(correlation);
map_refined = correlation(ceil(size_ref(1)/2):x -
ceil(size_ref(1)/2), ceil(size_ref(2)/2):y -
ceil(size_ref(2)/2));

%Find the points of maximum correlation
```

```
[locx, locy] = find(map_refined == max_val);


end
```

Approach 2

```
function stopfind(stopcolor,name)
tic

stopcolor = imread(stopcolor);                %
import the picture
stopcolor = imresize(stopcolor,[500,NaN]);  %
resize the picture
stop = rgb2gray(stopcolor);                   % make
it grayscale

stop = edge(stop);
% edge detector to get rid of extraneous elements
and convert to 1's and 0's

stop = bwareaopen(stop,30);                   % get
rid of all blobs too small to be important

bounds = regionprops(stop, 'boundingbox');      %
for all regions, find their bounds
centers = regionprops(stop, 'centroid');    % for
all regions, find their centre of mass
areas = regionprops(stop, 'area');

numBlobs = length(bounds);                    % find
the number of retions

regions = zeros(numBlobs,4);                  %
create a matrix to hold all the region data
height = zeros(numBlobs,1);
centerpts = zeros(numBlobs,2);
```

```
area = zeros(numBlobs,1);
for n = 1:numBlobs                                   %
    regions(n,:) = bounds(n).BoundingBox;   %
populate the matrix for each region
    height(n) = regions(n,4);
    centerpts(n,:) = centers(n).Centroid;   %
populate the center matrix for each blob
    area(n,:) = areas(n).Area;
end                                          %

numsamecenters=1;
for i = 1:numBlobs
    for j = i+1:numBlobs
            if(abs(centerpts(i,1)-
centerpts(j,1))+abs(centerpts(i,2)-centerpts(j,2))
10)
            %add a less than symbol in front of
the 10 [cnx would not let me import the actual
line]
            samecenters(numsamecenters) = i;
            numsamecenters=numsamecenters+1;
        end
    end
end

figure
axis ij
imshow(stopcolor)
hold on
title(name);
for n = 1:numsamecenters-1
    samecenter =
floor(centerpts(samecenters(n),:));
    abovept = [samecenter(1),floor(samecenter(2)-
height(samecenters(n)))];
    belowpt =
[samecenter(1),floor(samecenter(2)+height(samecent
ers(n)))];
    rightpt = [samecenter(1)-
floor(regions(samecenters(n),3)/3),samecenter(2)];
    left_pt =
[samecenter(1)+floor(regions(samecenters(n),3)/3),
samecenter(2)];
    if
checkColor2(abovept(1),abovept(2),stopcolor)
        if
checkColor2(belowpt(1),belowpt(2),stopcolor)
            plot(samecenter(1)-
regions(samecenters(n),3)/2,samecenter(2),'go',...

'markersize',4.5*regions(samecenters(n),3))
        end
    end

end

hold off

toc
end
```

Approach 3

```
function findAll

% Import the Templates
%--------------------------------------------------
-------------------------
S = imread('Templates/S.jpg');
% S template
T = imread('Templates/T.jpg');
% T template
O = imread('Templates/O.jpg');
% O template
```

```
P = imread('Templates/P.jpg');
% P template
STOP_W = imread('Templates/STOP_W.jpg');
% stop sign with white background
STOP_B = imread('Templates/STOP_B.jpg');
% stop sign with black background
SPEED = imread('Templates/SPEED.jpg');
% speed limit sign
ONEWAY = imread('Templates/ONEWAY.jpg');
% one way sign
DNE = imread('Templates/DNE.jpg');
% do not enter sign
%--------------------------------------------------
-------------------------

for k = 1:3
    sign =
imread(strcat('Do_Not_Enter/',num2str(k),'.jpg'));
% import the picture

    sign1 = rgb2gray(sign);
% make it grayscale
    sign1 = edge(sign1);
% edge detector to get rid of extraneous elements
and convert to 1's and 0's
    sign1 = bwareaopen(sign1,30);
% get rid of all blobs too small to be important
    bounds = regionprops(sign1, 'boundingbox');
% for all regions, find their bounds
    numBlobs = size(bounds);
% find the number of regions
    numBlobs = numBlobs(1);
% make that number a handy variable

    regions = zeros(numBlobs,4);                    %
create a matrix to hold all the region data
    for n = 1:numBlobs                              %
```

```
        regions(n,:) = bounds(n).BoundingBox;   %
populate the matrix for each region
    end

    checkS = zeros(1,numBlobs);
% matrix holding the correlations of each blob
with the S template
    checkT = zeros(1,numBlobs);
% ditto for the T template
    checkO = zeros(1,numBlobs);
% ditto for the O template
    checkP = zeros(1,numBlobs);
% ditto for the P template
    checkSTOP_W = zeros(1,numBlobs);
% ditto for the white stop sign template
    checkSTOP_B = zeros(1,numBlobs);
% ditto for the black stop sign template
    checkSPEED = zeros(1,numBlobs);
% ditto for the speed limit template
    checkONEWAY = zeros(1,numBlobs);
% ditto for the one way template
    checkDNE = zeros(1,numBlobs);
% ditto for the do not enter template

    for n = 1:numBlobs
        cropped = imcrop(sign, regions(n,:));
% choose one of the regions and crop it
        cropped = imresize(cropped, [57 35]);
% resize the image for correlation testing

        checkR = corr2(cropped(:,:,1), S(:,:,1));
% check red correlation between cropped image and
S template
        checkG = corr2(cropped(:,:,2), S(:,:,2));
% check green correlation between cropped image
and S template
        checkB = corr2(cropped(:,:,3), S(:,:,3));
```

```
% check blue correlation between cropped image and
S template
        checkS(1,n) = (checkR + checkG +
checkB)/3;     % sum the correlations

        checkR = corr2(cropped(:,:,1), T(:,:,1));
% red for T template
        checkG = corr2(cropped(:,:,2), T(:,:,2));
% green for T template
        checkB = corr2(cropped(:,:,3), T(:,:,3));
% blue for T template
        checkT(1,n) = (checkR + checkG +
checkB)/3;     % sum the correlations

        checkR = corr2(cropped(:,:,1), O(:,:,1));
% red for O template
        checkG = corr2(cropped(:,:,2), O(:,:,2));
% green for O template
        checkB = corr2(cropped(:,:,3), O(:,:,3));
% blue for O template
        checkO(1,n) = (checkR + checkG +
checkB)/3;     % sum the correlations

        checkR = corr2(cropped(:,:,1), P(:,:,1));
% red for P template
        checkG = corr2(cropped(:,:,2), P(:,:,2));
% green for P template
        checkB = corr2(cropped(:,:,3), P(:,:,3));
% blue for S template
        checkP(1,n) = (checkR + checkG +
checkB)/3;     % sum the correlations

        checkR = corr2(cropped(:,:,1),
STOP_W(:,:,1));        % red for white stop sign
template
        checkG = corr2(cropped(:,:,2),
STOP_W(:,:,2));        % green for white stop sign
```

```
template
        checkB = corr2(cropped(:,:,3),
STOP_W(:,:,3));        % blue for white stop sign
template
        checkSTOP_W(1,n) = (checkR + checkG +
checkB)/3;      % sum the correlations

        checkR = corr2(cropped(:,:,1),
STOP_B(:,:,1));        % red for black stop sign
template
        checkG = corr2(cropped(:,:,2),
STOP_B(:,:,2));        % green for black stop sign
template
        checkB = corr2(cropped(:,:,3),
STOP_B(:,:,3));        % blue for black stop sign
template
        checkSTOP_B(1,n) = (checkR + checkG +
checkB)/3;      % sum the correlations

        checkR = corr2(cropped(:,:,1),
SPEED(:,:,1));       % red for speed limit
template
        checkG = corr2(cropped(:,:,2),
SPEED(:,:,2));       % green for speed limlit
template
        checkB = corr2(cropped(:,:,3),
SPEED(:,:,3));       % blue for speed limit
template
        checkSPEED(1,n) = (checkR + checkG +
checkB)/3;     % sum the correlations

        checkR = corr2(cropped(:,:,1),
ONEWAY(:,:,1));       % red for one way template
        checkG = corr2(cropped(:,:,2),
ONEWAY(:,:,2));       % green for one way template
        checkB = corr2(cropped(:,:,3),
ONEWAY(:,:,3));       % blue for one way template
```

```
        checkONEWAY(1,n) = (checkR + checkG +
checkB)/3;     % sum the correlations

        checkR = corr2(cropped(:,:,1),
DNE(:,:,1));       % red for one way template
        checkG = corr2(cropped(:,:,2),
DNE(:,:,2));       % green for one way template
        checkB = corr2(cropped(:,:,3),
DNE(:,:,3));       % blue for one way template
        checkDNE(1,n) = (checkR + checkG +
checkB)/3;     % sum the correlations
    end

    % Nice display (uncomment all)
    %fprintf(strcat(num2str(k),':\n'));
    %fprintf(strcat('S: ', num2str(max(checkS)),
'\n'));
    %fprintf(strcat('T: ', num2str(max(checkT)),
'\n'));
    %fprintf(strcat('O: ', num2str(max(checkO)),
'\n'));
    %fprintf(strcat('P: ', num2str(max(checkP)),
'\n'));
    %fprintf(strcat('STOP_W: ',
num2str(max(checkSTOP_W)), '\n'));
    %fprintf(strcat('STOP_B: ',
num2str(max(checkSTOP_B)), '\n'));
    %fprintf(strcat('SPEED: ',
num2str(max(checkSPEED)), '\n'));
    %fprintf(strcat('ONEWAY: ',
num2str(max(checkONEWAY)), '\n'));
    %fprintf(strcat('DNE: ',
num2str(max(checkDNE)), '\n\n'));

    % Useful display for copying and pasting into
a table (uncomment one line at a time)
    %fprintf(strcat(num2str(max(checkS)), '\n'));
    %fprintf(strcat(num2str(max(checkT)), '\n'));
    %fprintf(strcat(num2str(max(checkO)), '\n'));
    %fprintf(strcat(num2str(max(checkP)), '\n'));
    %fprintf(strcat(num2str(max(checkSTOP_W)),
'\n'));
    %fprintf(strcat(num2str(max(checkSTOP_B)),
'\n'));
    %fprintf(strcat(num2str(max(checkSPEED)),
'\n'));
    %fprintf(strcat(num2str(max(checkONEWAY)),
'\n'));
    %fprintf(strcat(num2str(max(checkDNE)),
'\n'));

end

end
```